Learning Behaviors Through Demonstration:
Artificial Intelligence for Non-Player Characters in an Interactive Drama

Thomas C Amundsen III

A Thesis Submitted in Partial Requirements for the
Research Option in the College of Computing, Bachelor's of Science in Computer Science

Faculty Advisor: Dr. Ashwin Ram

Second Reader: Manish Mehta

Key Words: behavior demonstration, interactive drama, case-based reasoning, Unreal Tournament 2003, video game, murder mystery, artificial intelligence

Date: 4 December 2007

Abstract

In both the game industry and the academic community, there have not been many attempts to create complete interactive drama systems. I am working with a group that will be the first to undertake this daunting challenge, and one of the most important components in such a system is artificial intelligence (AI) to control the behaviors of non-player characters. Traditional approaches to designing the AI for any kind of interactive game involves designing characters who follow scripted behaviors. This method is cumbersome and amounts to gameplay that is repetitive and thus inhuman. This paper will describe my attempt to design a system that will allow an expert user to demonstrate behaviors to the system, which the system will use to learn how to behave on its own using case-based reasoning. The end result of this work will not only benefit interactive dramas but may help in the design of game AI for other genres of video games or simulations.

Introduction

Interactive dramas have been a part of American culture for over two decades. About twenty years ago, a trend called 'Gamebooks' was developed to allow readers to make choices that would affect the outcome of a story. While reading one of these books, the reader is presented with a decision that could result in multiple outcomes. Depending on which choice was made, the reader would be instructed to turn to a different page number to continue the story. This idea has permeated today's video games, and we can see many 'Role Playing Games' that give players nearly limitless freedom in choosing their actions. However, none of these games allows players to have interesting conversations with other characters in the game. In light of this shortcoming, there have been a few recent attempts in the research community to design interactive drama systems that allow a person to play a game where they are able to talk to characters and have a meaningful impact on the outcome of the plot.

The first successful attempt at creating an interactive drama was a game called Façade, developed by Michael Mateas at Georgia Tech. In his paper, 'Façade: An Experiment in Building a Fully Interactive Drama,' Mateas describes how the current game industry is focused on allowing players to *physically* interact but not communicate with natural language. He proposes that creating games with open-ended physical

manipulation of the world will never lead to a fully interactive system since the narrative will always be rigidly structured. Instead, a system that allows players to interactively change the structure of the narrative will ultimately lead to an interactive story, which "has been a Holy Grail of game design since the advent of computer games." [4]

In the game Façade, one is able to play the long time friend of Trip and Grace. The player may select a name and gender and can choose from a set of physical interactions or say anything to either character by using a keyboard. When the game starts, one arrives at Trip and Grace's apartment for a little get together with the couple. As the 'game' progresses, one finds out that their marriage is not going well and is forced to help them try to resolve their problems. The actions a player performs can lead to many different story endings, and the game encourages users to replay it many times. In fact, one of the main goals of the project was to make players feel like they have 'exhausted' the possible plotlines only after playing the game six or seven times. This open-endedness is facilitated by the fact that there are no main branch points in the plot that a player is able to identify. Instead, the player is given a small number of subtle behavioral choices that will affect the direction of the plot. [4]

Trip and Grace's behaviors in Façade were authored using 'A Behavior Language' (ABL), which is described in Mateas' paper, 'A Behavior Language for Story-Based Believable Agents.' ABL allows programmers to create a set of behaviors that define how a character will behave in a game or simulation. It uses the concept of preconditions to determine which action to select when faced with a decision. (For instance, if I want to go inside a room in a house, I must first have the key to open its door.) Using ABL, game developers can develop a robust simulation with interactive characters; however, creating enough behaviors to accomplish this goal takes time. Mateas estimated that a total of 20 man-years of behavior authoring was involved in the creation of Façade. Obviously, this is not a feasible method of behavior authoring, so my thesis work is an attempt to design a system that will allow game developers to generate agent behaviors in much less time than it would take to write them by hand. I plan to design a system that will use Case-Based Reasoning (CBR) to learn character behaviors through expert demonstration and apply these acquired behaviors during runtime execution. [5]

In order to understand how I will extract and apply these behaviors acquired during expert demonstration, it would be helpful to first understand what case-based reasoning is. Different from most other AI techniques, CBR relies on learning specific information in the form of cases instead of devising a general strategy to approach problems. In general, we can say that CBR is a programming method that solves problems by remembering a situation in the past and applying knowledge learned to new situations. As a direct result of this style, CBR "is an approach to incremental, sustained learning, since a new experience is retained each time a problem has been solved, making it immediately available for future problems." [1] CBR is based on psychological studies which have shown that humans use similar methods in their decision-making process. People often remember how they 'solved a problem' when they must decide to react to a new situation; this method of learning can be easily observed in children during their early developmental years. Because CBR has been inspired largely by human psychology, which allows for many interpretations, there are a vast number of methods used in intelligent systems that are known as 'Case-Based Reasoning.' [1]

At the very lowest level, all CBR techniques involve the organization, retrieval, and use of information. There are many different ways that information can be organized. Systems may use one of the following methods of organization: store concrete experiences, generalize information, index information based on some identifier which can be organized in many different kinds of structures, create 'knowledge units' which are possibly split into subunits, or a number of other various techniques. When retrieving information, cases may be matched only by visible or syntactic similarity, or they might be guided by a general or specific domain of knowledge. Sometimes solutions to problems are applied directly to new ones; at other times they are manipulated to fit each new situation. [1]

The authors, Aamodt Agnar and Plaza Enric, have devised a framework that describes CBR: the process model of the CBR cycle and a 'task-method' structure. The case-based reasoning cycle can be defined by four general stages (depicted in Figure 1): retrieve, reuse, revise, retain. In the retrieve stage, the system will start with the problem partially defined and select the existing cases that are most relevant to the problem. When a new problem is initially described, it defines a new case. This stage typically consists of

four sub-phases: identify features, initially match, search, and select. While identifying features, the system will extract 'problem descriptors' that hopefully describe each case. It will then match these descriptors to the new case depending on how relevant they are. Finally, search and select will pick which cases have the best matches. [1]

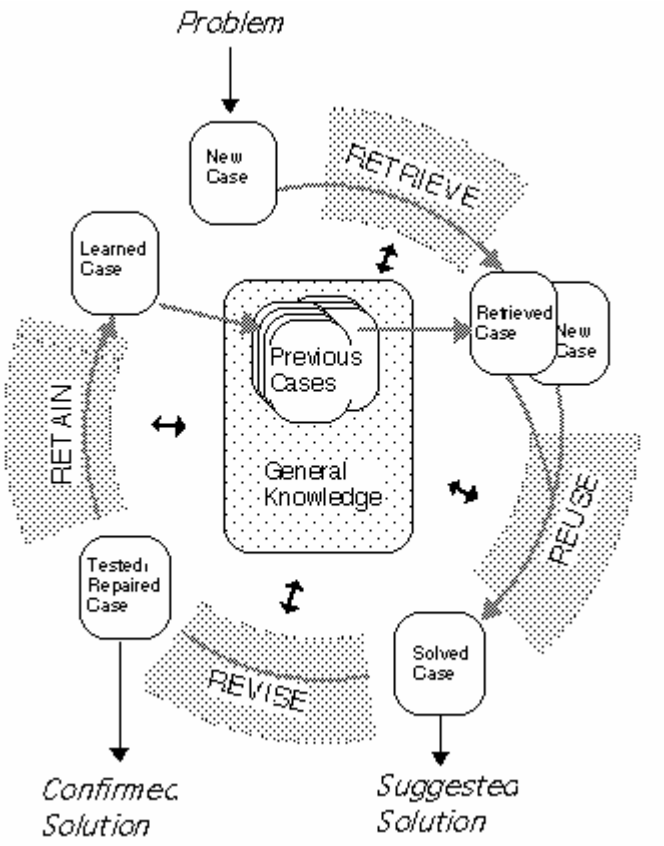

Figure 1.
Illustration of the case-based reasoning cycle that presents the order of the four stages involved. Reproduced with permission from [1].

During the reuse stage, the system adapts the cases selected in the retrieve stage so that they are applicable to the problem. The system will combine information from the new case with the retrieved previous cases and attempt to formulate a solution case. In this stage, we must consider the differences between the previous cases and the new case and what information can be used from the previous case to solve the new problem. In complicated systems, this process is typically accomplished in two steps: copy and adapt. Some information from the matching cases will be copied into the new case, but if there are significant differences found, certain parts of a case may need to be adapted before they can be incorporated into the solution. [1]

In the revise stage, the solved case is tested until it succeeds in accomplishing the goal. In order to do this, one must first evaluate the solution case and then reformulate it if it fails to solve the problem. Testing the solution can follow two different courses of action. The first solution might solve the problem, and we can learn directly from it; or we will learn from its failure, redesign the solution and test it again. Evaluating the system necessarily involves execution of a process outside of the CBR cycle, so there is some lag time until we can receive feedback. Repairing the solution in the case of failure can be very difficult and is approached in different ways. [1]

Finally, in the retain step of the CBR cycle, the system can either add this newly solved case into its set of cases, or it can discard the solution if it determines it is unfit for reuse. Sometimes the system will simply keep some of the information from the new case and combine it with an old case. Because the system is adding new information into its set of existing cases, this is typically known as the 'learning' phase. Learning is typically accomplished in three steps: extract, index, and integrate. In extraction, we may either build an entirely new case or just generalize an old case to account for the new solution. Problem descriptors found in the retrieve stage should be analyzed to determine what information should be retained. After deciding what information to keep, there is a much more difficult problem of how to index the information. This step in the process is a big issue in Case-Based Reasoning and is described as being a 'knowledge acquisition problem,' which involves deciding what kind of indices to use and how to structure those indices. The final phase of the retain step is integration, where we actually update the knowledge base with our newly acquired information. If we have not created a new case and index structure, this is actually the only step in this final phase. If we have created a new case with its own index structure, it is very important that we 'tune' the indices and assign weights to each so that we can more effectively search them in the future. [1]

Case-Based Reasoning can be used to generate behaviors for virtual characters in interactive games. The most relevant paper in this research area is 'Learning from Demonstration and Case-Based Planning for Real-Time Strategy Games' written by Santi Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. [6] This group of scientists performs their research at Georgia Tech's Cognitive Computing Lab, where I am doing my thesis work. In their paper, they present a system that extracts behavioral knowledge

from demonstrations in 'Warcraft II – Wargus,' stores information from the demonstrations in cases, and retrieves appropriate behaviors at runtime to fit the situation at hand. The motivation behind their research is that they believe traditional Artificial Intelligence (AI) techniques fail to play at the human level and require an exceedingly large amount of effort to design. They describe how the vast size of search spaces in real-time strategy (RTS) games inhibits the use of traditional search-based techniques and require game developers to hand-code behaviors, which is a tedious task. [6]

The techniques used by Ontañón et al. are applicable to my research because the project I am working on previously used hand-written behaviors to control the agents in our system. In this project, characters in the interactive 'Murder Mystery drama' are agents who have specific personalities and must be able to convince a person playing the game of their intelligence and must act as human-like as possible. Our previous method, writing behaviors by hand, took five people four months to program a scene that was less than a minute long. At that rate, it would be impossible for us to complete the project, and our efforts would have been futile. My plan is to design a tool that will rely on the principles presented in the paper by Ontañón et al. to generate behaviors for the Mystery Mansion interactive drama and save our team time and effort. [6]

In addition to learning behaviors through demonstration, the paper by Ontañón et al. presents an integrated architecture for case-based planning and execution. It is described as an 'integrated architecture' because plan retrieval, composition, adaptation, and execution are synthesized in a stand-alone functional system. The system maintains a list of open goals, starting with the goal of winning the game, and chooses the behavior that is most likely to achieve each of these goals. After choosing appropriate behaviors, it adds them to the current plan and adapts them to the current gamestate. Each of these behaviors is monitored for success or failure; if a behavior fails to achieve its goal, the system searches for a better behavior that is more likely to result in success. This process allows the system to use behaviors extracted from an expert demonstration and use them in the game to provide intelligent, non-repetitive gameplay. [6]

The paper by Ontañón et al. also describes many technical details of their system to allow other scientists to replicate their work or apply it in a different way, which is the aim of this thesis. In order to extract information from user demonstrations, Ontañón et

al. allow a person to play the game, and their system creates a trace of each behavior they perform. A trace is representation of all the actions a user has performed in a simulation, where every entry in the trace indicates the cycle when the action occurred, the name of the player who executed the action, and the action that was executed. After the game is played, users annotate the trace with information about what goals they were trying to accomplish while performing each action. The scientists use these annotations to create a set of cases which are stored as triples containing information about the situation, the goal, and the behavior performed. Behaviors are executed by a separate engine, which consists of two modules. A real-time plan expansion and execution module keeps an 'execution tree,' which keeps track of all open goals and their associated behaviors. There is also a behavior generation module that is queried for each open goal. Upon receipt of a query, this module selects the case that is most likely to close that goal with a success. An illustration of how these components interact is shown in Figure 2. [6]

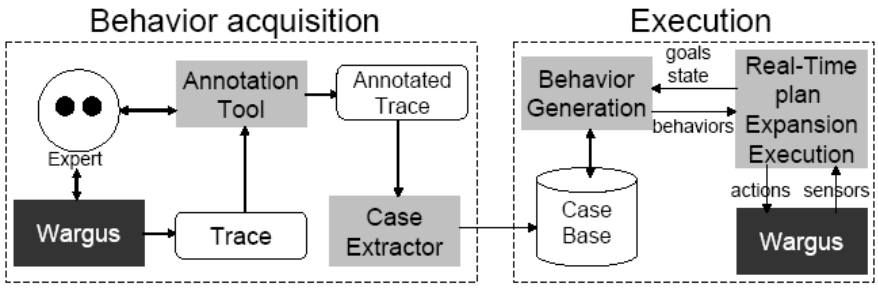

Figure 2.
Diagram of the behavior generation and execution system used by Ontañón et al. Reproduced with permission from [6].

The traces used in the group's behavior extraction will be very important for my work, so I will now describe exactly how Ontañón et al. trace and annotate a user's behaviors. Each action must be associated with a goal, which is defined by a name and a set of parameters. Each type of goal has a different set of parameters; some examples from the system designed by Ontañón et al. are WinWargus(player), which indicates that the expert's goal is to win the game; DefeatPlayer(player, opponent) which indicates that the expert's goal is to defeat a particular opponent; and SetupResourceInfrastructure(player, peasants, farms), which defines a goal where the expert wants to set up a resource infrastructure that involves a certain set of peasants and farms. These annotations result in a trace that gives us ample information about each

action performed: the cycle number, the player, the behavior demonstrated, and the annotation which describes the expert's intentions. An example of traces produced by Ontañón et al. is shown in Figure 3. [6]

Cycle	Player	Action	Annotation
8	1	Build(2,"pig-farm",26,20)	-
137	0	Build(5,"farm",4,22)	SetupResourceInfrastructure(0,5,2) WinWargus(0)
638	1	Train(4,"peon")	-
638	1	Build(2,"troll-lumber-mill",22,20)	-
798	0	Train(3,"peasant")	SetupResourceInfrastructure(0,5,2) WinWargus(0)
878	1	Train(4,"peon")	-
878	1	Resource(10,5)	-
897	0	Resource(5,0)	SetupResourceInfrastructure(0,5,2) WinWargus(0)
…	…	…	…

Figure 3.
Some examples of traces and annotations used in the system created by Ontañón et al. Reproduced with permission from [6].

The paper written by Ontañón et al. was inspired largely by a text entitled, 'Watch What I Do: Programming by Demonstration.' This book was written in 1993 and is composed of a series of articles that discuss the concept of 'programming by demonstration.' One of the articles in this collection relevant to my research is called 'Tinker: A Programming by Demonstration System for Beginner Programmers' by Henry Lieberman. This article begins by insisting on the effectiveness of learning by demonstration by noting that most (human) teachers often use concrete examples to teach students new concepts instead of teaching them abstract rules that they will likely forget. The author proposes the idea that if this method of teaching works well for humans, then it must also work well for computers; after all, computers are only an extension of our own minds. [6]

Using this idea, Lieberman designed a system called Tinker that is able to 'learn' from examples that the programmer demonstrates. The programmer must let Tinker know which aspects of the example are important to allow Tinker to build its knowledge by incrementally learning different examples. In order to describe how Tinker works, the author gives an example of how it functions in the block world. The block world is a cliché example in computer science where there are a set of blocks on a table. With each

move, one can put a block on top of another block or take a block off of another. The only limitations are that if, for example (as depicted in Figure 4), block Z is on top of block X, then block Y cannot be placed on top of block X, or vice versa. [2]

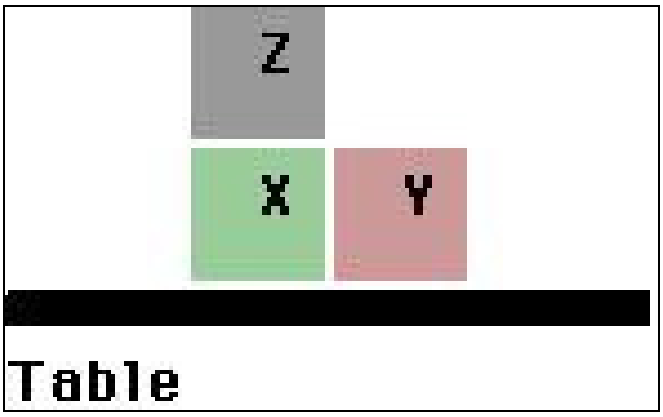

Figure 4.
This diagram depicts a situation in 'The Block World' where it is not possible to place block Y on top of block X, or vice versa. Reproduced with permission from [2].

When programmers want Tinker to respond to a situation in different ways, they must demonstrate that there are multiple courses of action that can be taken. However, for each possible course of action, the programmer must provide Tinker with a conditional test that will let it know when to take which alternative. Actions are generalized in the system as procedures (remembered sets of actions) that are given arguments that are able to behave differently each time they are executed. Tinker uses the List Processing (LISP) programming language and allows users to write their own LISP functions and assign them to particular actions that define the result of executing the function. Going back to the 'block world' example, a programmer can demonstrate the action – "Put block X on top of block Y." Tinker does not remember the names of the blocks, but generalizes the action and remembers that the programmer has put one block on top of the other. In this case, Tinker simply learns one piece of LISP code, "(Move-Block From To)." This line of code is a generalization of 'put block X on top of Y' to the simpler form of moving a block from one location to another. This example is useful, but it would not be possible to perform more complex tasks if the system can only learn simple behaviors like this. [2]

In the block world domain, a very important concept one must understand is the existence of obstacles. Lieberman has designed a LISP member function for each block called 'above' that returns the block on top of it or NIL if there is none. When the

programmer demonstrates this behavior, Tinker learns the general function call, "(Above From)". Using the two actions that I have described, Tinker has all the basic concepts it needs. It is able to manipulate blocks in order to accomplish a specified goal by searching through its set of demonstrated actions, find out which demonstration is most similar to what it needs to get done, and manipulate the behavior for immediate use. However, if the programmer does not provide Tinker with 'good' examples, the system may not be able to achieve its goal. [2]

One attribute of a 'good' set of examples is that they should be in sequential order of increasing complexity. Tinker is able to understand recursive procedures (procedures that make a call to themselves with modified input parameters), but the programmer cannot start the sequence of examples with such a complicated action. One interesting aspect of the system is that the sequence of examples can actually start with simple actions that are actually 'wrong' but allow the system to be able to learn more complex examples using these definitions. While developing my system, I must also follow this same principle and demonstrate behaviors in increasing complexity so that it will be able to learn complicated behaviors after some incremental learning has taken place. [2]

Description of Domain

This thesis is one component of a system that hopes to fulfill the goal of realizing a fully interactive narrative in a 'murder mystery' setting. The entire system will include the following components: a drama manager that will make important decisions that influence the plot, such as making one of the NPCs kill another; a natural language processing (NLP) component that will analyze the text input by the user and represent it in a meaningful way so that the NPCs will be able to respond appropriately; and a natural language generation (NLG) component that will generate dialogs for the NPCs to communicate with the other players in the game.

The project is entitled 'Mystery Mansion,' and the narrative takes place in the 'Hill Top Manor' owned by Paul Andersen, during the 18th century in England. Paul is a wealthy doctor and has just bought the house as a gift for his daughter, Mary. The Andersens are celebrating the purchase of their new house and have invited a few of their neighbors over for dinner and cocktails. Their guest list includes the following people:

Max Casanova, Amanda Levi, and Manuel Sharma. On the night of their party, a terrible snow storm occurs, and the guests are forced to stay the night in the mansion.

My behavior demonstration tool is being designed to learn behaviors for the characters in Mystery Mansion, so it is very important to understand the storyline associated with this interactive drama. Hill Top Manor was previously owned by the O'Sullivan family, and the McCarthy family has been serving as butlers in their home for many generations. The O'Sullivans treated the McCarthys with great respect, and they almost seemed to be one large family. Jeeves McCarthy was the only remaining member of the butler family after the last O'Sullivan had died, and Paul agreed to hire him so that Jeeves would not lose his job. Jeeves still feels more loyal to the O'Sullivans than to the Andersens, and because of this there is much tension between Jeeves, Paul, and Mary.

Paul's daughter, Mary, has been dating a chauvinistic, British-Italian man named Max Casanova. Max's family has been involved in the Mafia for many generations, and he was no exception to that rule. Since a very early age, Max has been involved in many illegal activities such as robbing, bootlegging, and murder. Because of Max's past, Paul does not approve of Mary's relationship with him and is willing to try anything to break the two apart. Paul has not recognized the softer side of Max that Mary has fallen in love with. Mary constantly tries to tell her father how Max has changed, but is never able to win Paul's approval.

To make the story even more interesting, one of Paul's neighbors, Amanda Levi, is invited to the party. Amanda is Max's ex-girlfriend who has never gotten over their breakup. She believes that Mary is not good enough for Max and is always trying to break up their relationship. Several times throughout the story, Amanda tries to flirt with Max to make Mary upset. Amanda causes stress in Mary and Max's relationship and heats up the drama so that the characters will become even more upset with each other.

The only remaining character in the story is Manuel Sharma. Manuel is a civil engineer who happens to live near Mary's new house. He is the only character who does not have a prior relationship with anyone else in the story. Because of this, Manuel is a neutral character who does not have much to talk about besides identifying flaws in the construction of the house. All the other characters in the story have tense relationships

with each other, and depending on how one plays the game, these tensions might become extremely heated and cause one character to murder another.

The drama consists of four scenes: the cocktail party, the dinner, the murder, and the aftermath. The drama manager will constantly monitor the actions of the game for specific plot points that determine when to transition between scenes. During the cocktail party, the characters mingle with each other and those who have not met introduce themselves. During this scene, Paul rudely orders Jeeves to prepare drinks for the guests, which makes Jeeves even angrier with his new master. During this scene, Paul lets the guests know that they will stay the night because of the terrible storm. However, there are not enough rooms for everyone, so some people will have to share. This predicament causes the first real conflicts between the characters, and there are several arguments about who might sleep in the same rooms. The love triangle between Max, Mary, and Amanda will obviously cause some concern among themselves and Mary's father.

Towards the end of the first scene, Mary and Max have a discussion in the kitchen about their relationship. The couple has been dating for a long time, and they are considering getting engaged. Either Max or Mary will bring up the topic of engagement. Depending on their previous interactions in the drama, there might be a disagreement of whether or not to announce their commitment to the others at the party. Some obvious factors that might cause disagreement are Paul's disapproval of Mary's relationship with Max and the presence of Max's old girlfriend, Amanda. It is important that the experts using my tool demonstrate these situations. The knowledgebase will need to contain cases for every aspect of the plot so that the characters will be able to repeat these behaviors in the game.

After the cocktail party, Paul invites the guests into the dining room to the dinner that Jeeves has prepared. Jeeves has cooked a traditional British dish that consists mainly of sausage. He was unaware that Amanda is Jewish and is unable to eat pork and that Manuel is a Hindu and complete vegetarian. Paul orders Jeeves to prepare a replacement dish for the two guests who are unable to eat the main course. This order frustrates Jeeves, because he knows that there are plenty of side dishes that they could eat that do not contain meat.

After everyone is well into their dinners, Max might decide to announce his engagement with Mary. The announcement will cause a lot of argument from both Paul and Amanda. Since Paul does not want Mary to be involved with Max at all, he will be angered by this announcement. Depending on Paul's current mood in the game, he might become enraged and cause a big ruckus. However, if Max has acted friendly to Paul throughout the game, Paul might keep his thoughts to himself. Amanda will also be upset about this announcement since she wants to have Max for herself. As a result, she might try to talk Max out of the engagement or flirt with him to make Mary upset.

Clearly there are many sensitive relationships in this story, and there may be other events that occur that create a strain on the characters. By the time the second scene ends and the characters go to sleep for the night, there will likely be an extreme conflict between two of the characters, and one of them will murder the other. When everyone wakes up the next morning, they will discover the dead body and try to identify the perpetrator. The person playing the game will be able to talk to anyone in the game during the investigation. At the conclusion, the murderer should be identified, and the player will want to replay the game to see what happens differently in changed situations or how a different impact on the game can be obtained by controlling a different character.

ABLUT

Our team decided to implement the interactive drama, 'Mystery Mansion,' by using Unreal Tournament 2004 (UT). We have artists creating models for the characters and the mansion which we will import into a UT map. The Unreal Engine gives us the ability to launch a first-person game, where a player can navigate with a mouse and keyboard, without doing any work. This leaves us with the responsibility of adding intelligent characters into the game and providing the user with an interface to interact with the system.

At the outset of the project, our team used ABL to create character behaviors. This method required us to create ABL files for each character (refer to Figure 5 for an example of an ABL behavior). Characters' ABL files define exactly how they behave in a sequential manner. When a specific precondition is met, such as one character bumps into another, the ABL file makes sure that the character will behave in a predefined way. In

order for the characters to 'hear' what the other people are saying, we had to create a 'Global Memory' to keep track of important data such as the verbal output of every character. The process of defining every possible precondition and associated response, along with our method of retrieving data from global memory made this approach very tedious and time consuming. When the game was played and the characters behaved according to their ABL behaviors, there were constantly problems in the definition of preconditions and sequence of actions. That resulted in a lot of debugging, and our group of five programmers only managed to complete the initial character introductions after programming for about four months.

```
/**
* Controls the behavior of this character as they enter the mansion at the
* beginning of the game.
* pre: Waits for Max to enter the mansion
* post: Paul enters the mansion, is greeted by Jeeves, and responds.
**/
sequential behavior EnterMansion() {
	double x,y;
	int plotID = PlotPointWME.READY_FOR_GUEST_TO_ENTER_HOUSE, speech;
	String message;
	with ( success_test { GlobalMemory (PlotPointWME participants == Max id == plotID)}) wait;

	//This mental act sets the values for the goal location. Should use
	//important locations when it is completed
	mental_act {
		//System.out.println("Paul - EnterMansion fired");
		BehavingEntity.getBehavingEntity().deleteAllWMEClass("PathWME");
		x = 192.0; //-41
		y= -508.0; //-81
	}

	act pathplan(x,y,z);
	subgoal WalkPath();
	//setup the PlotPointWME that informs Jeeves that Amanda has entered the house
	mental_act {
		PlotPointWME plot = new PlotPointWME(PlotPointWME.GUEST_ENTERS_HOUSE);
		plot.setParticipants(Paul);
		WorkingMemory.addWME("GlobalMemory", plot);
	}

	with ( success_test { GlobalMemory (VerbalOutputWME sender == Jeeves receiver == Paul
speechDelay :: speech)}) wait;
	with (priority 100) subgoal faceJeeves();
	mental_act { System.out.println("Paul - speechDelay: " + speech); }
	//respond to Jeeves
	mental_act {
		message = "Damn! My first day in this house and things are already going wrong!";
		VerbalOutputWME voWME = new VerbalOutputWME(Paul, Jeeves, message);
```

```
                WorkingMemory.addWME("GlobalMemory", voWME);
        }
        act sendglobalchatmessage(message);
}
```

Figure 5.
An example ABL behavior that defines how Paul acts after entering Hill Top Manor[2].

We soon learned that the ABL method was inefficient and that we would never finish the project if we continued down this path. Instead of using ABL, I have decided to create a system that will learn behaviors through expert demonstration. I am using ABLUT, the library that allowed ABL to interact with the Unreal Engine, without writing ABL behaviors. ABLUT is a library that creates a network connection with the Unreal Engine to allow programmers to spawn 'Pawns' into the current game and control their behavior (refer to Figure 6).

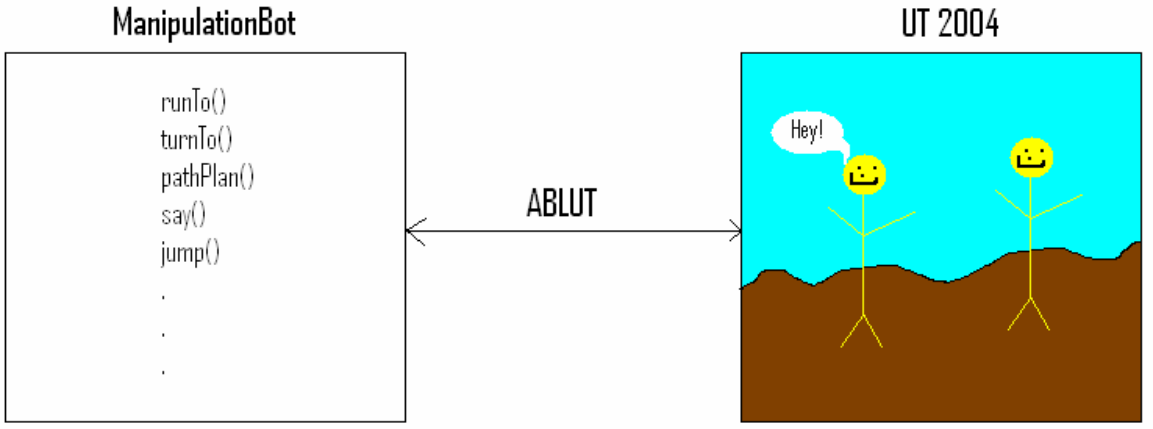

Figure 6.
This illustration shows the function of ABLUT which enables communication between my Java-side behavior demonstration code and the UT 2004 runtime environment.

I created a Java class called 'ManipulationBot' that owns two instances of objects from ABLUT: GamebotsClient and ProxyBot. GamebotsClient, written by Andrew N. Marshall, is a class that maintains the socket connection between Java and UT. [3] Its main functionality consists of the methods 'connect', 'disconnect', 'sendMessage,' and a few methods that receive and parse messages from UT. By using the GamebotsClient, one has all utilities taken care of in order to send and receive messages to and from UT.

ProxyBot, written by one of Michael Mateas' students in Georgia Tech's school of Literature, Communication, and Culture is the workhorse of the ManipulationBot

class. ProxyBot extends the Bot class, also written by Andrew N. Marshall, uses the GamebotsClient to send and receive messages from UT, ands does additional work to allow programmers to control their characters. ProxyBot has methods that allow a character to perform the following actions: run to a specific location, jump, turn, speak, and play both facial and skeletal animations. For these functions to work, ProxyBot sends a message to UT describing what action must be performed. UT parses this message and calls the corresponding method on its associated Pawn.

Some of ProxyBot's actions are much simpler than others. For instance, speaking only requires UT to call the function 'say(),' whereas running to a target requires the user to determine the exact three-dimensional coordinates of the destination and then call the 'runTo(double x, double y, double z, WalkTo Action)' method. However, if the user only calls 'runTo,' the character will run in a straight line from the current location to the destination. If there happens to be an obstacle in between these two points, the character will never reach the final destination, and it will be impossible to regain control over this character. In order to circumvent this problem, ProxyBot has a method called 'pathPlan(double x, double y, double z, PathPlan Action),' which returns a list of pathnode locations (which are described two paragraphs below) that can be traversed in order to reach the specified destination. This method is useful because it guarantees that the list of pathnodes will be reachable and that if the character traverses them in order they will reach the destination.

In addition to these physical actions, ProxyBot provides a set of sensors that allow programmers to access data gathered from UT. Some of the sensors that ProxyBot supports include AgentPosition, Bump, GlobalChatMessage, NavPoint, ObjectPosition, ObjectReachable, PlayerRotation, PlayerVelocity, and Self. These sensors can be thought of as devices mounted on a character that can detect the presence of whatever data needs to be sensed (see Figure 7).

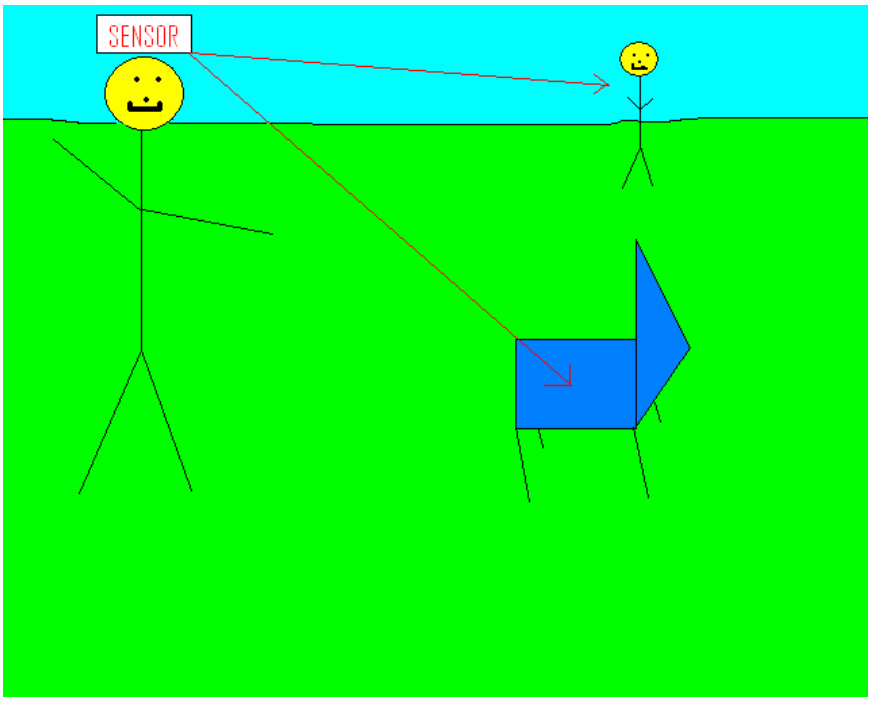

Figure 7.
This illustration depicts how sensors in ABLUT operate. There are separate sensors for each type of data that can measure values of different object attributes that exist in the character's field of vision.

The only sensor that is not self-explanatory is the 'NavPointSensor.' A 'NavPoint' is ABLUT's representation of a 'PathNode' in UT. In UT, there are artificial characters known as 'bots' with which players can compete. To allow these bots to navigate through maps, the game designers have created pathnodes that designate routes to be used for running through the map. When a bot wants to go from one location to another, it will calculate the shortest path that will take it from its current location to its destination. Since programmers using ABLUT would also like to use these pathnodes, there are 'NavPoint' and 'Path' working memory elements (WMEs).

"WMEs are like classes in an object oriented language; every WME has a type plus some number of typed fields which can take on values… WMEs are also the mechanism by which an agent becomes aware of sensed information." [5] Each sensor supported by ProxyBot stores information into a corresponding WME. These sensors are constantly retrieving data from UT, and anytime we want to check their values we simply call a method such as 'getAllAgentPositions()' which returns an array of 'AgentPositionWMEs.' As we will see later in this paper, WMEs are very important for

defining the current map and gamestate in case descriptions used for CBR-based behavior generation.

Methods

I have created a Java program that allows users to demonstrate behaviors in UT and outputs a log detailing every action the user has performed. After the log has been generated, the user will be able to annotate the log using another program which I have modified to handle my log file format. Once the user has started UT, they will run this program and see that a character is spawned into the game. This character is controlled by the previously discussed ManipulationBot class. The user observes the bot from a third person perspective and is able to control it using a Graphical User Interface (GUI). The GUI consists of a series of buttons and text fields that allow users to perform the following actions: speak, move forward, move backward, move left, move right, rotate, and play an animation (see Figure 8).

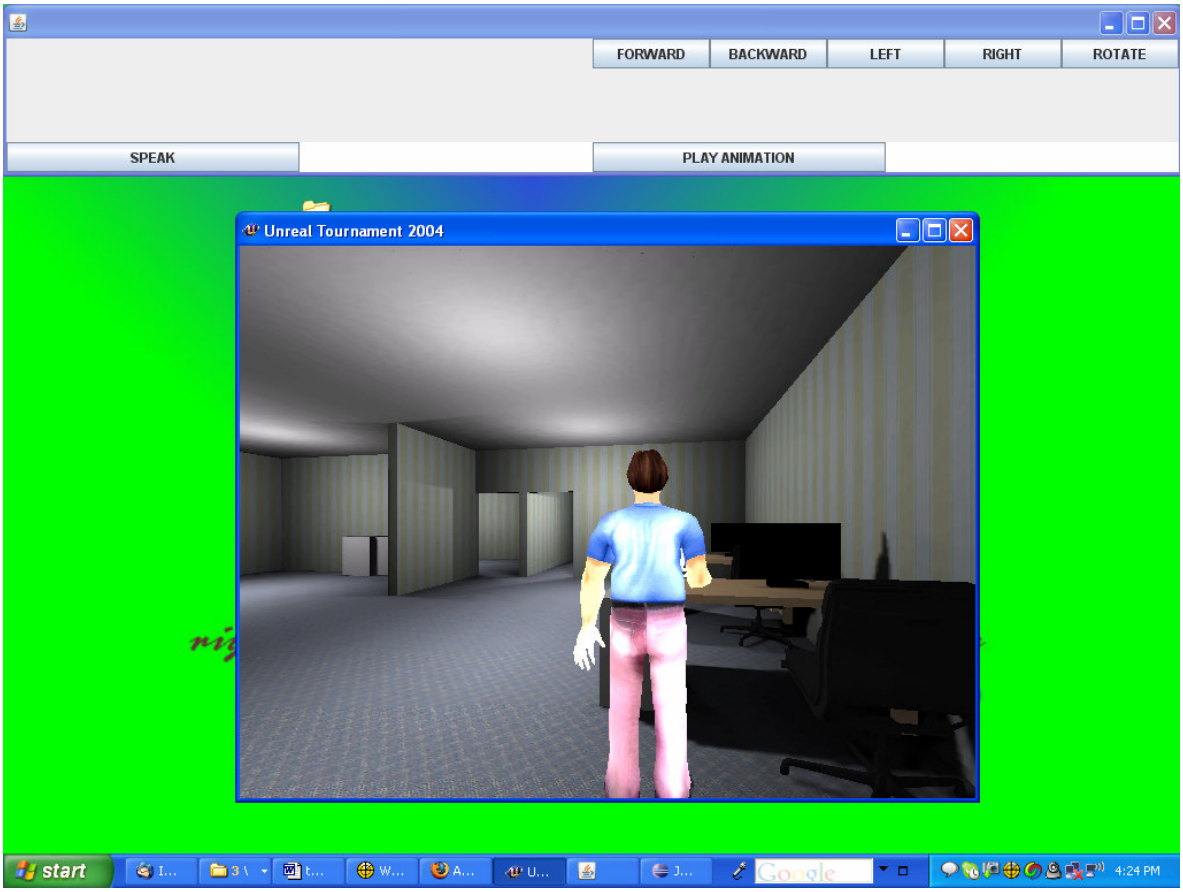

Figure 8.

Screenshot of the user interface and puppet character in Unreal Tournament 2004. An expert is able to demonstrate behaviors to the system by using the buttons seen in the interface at the top of the screen.

When a user demonstrates a new behavior, a description of the action and the current gamestate are written to a log file (see example in Figure 9). The log is written in XML and parameterizes the data in a machine-readable format. Each time an action is logged, the name of the action, the parameters associated with that action, the values attributed to those parameters, and the context in which the action was performed are written to the log file. A simple action like *jump* may not have any associated parameters, but a *walk* command might have parameters such as direction and distance.

```
<?xml version="1.0" encoding="UTF-8"?>
<ACTIONS>
    <ACTION>
        <PLAYER-NAME>Tracy</PLAYER-NAME>
        <NAME>SPEAK</NAME>
        <PARAMETERS>
            <MESSAGE>Hi, Tracy!</MESSAGE>
        </PARAMETERS>
        <CONTEXT>
            <MAP>
                <OBJECT>
                    <NAME>ProxyBot</NAME>
                    <POSITION>
                        <X>147.56</X>
                        <Y>-107.68</Y>
                        <Z>-1997.12</Z>
                    </POSITION>
                </OBJECT>
                <OBJECT>
                    <NAME>ProxyBot</NAME>
                    <POSITION>
                        <X>64.0</X>
                        <Y>-118.0</Y>
                        <Z>-1997.1</Z>
                    </POSITION>
                </OBJECT>
                <OBJECT>
                    <NAME>TOM</NAME>
                    <POSITION>
                        <X>-166.47</X>
                        <Y>47.29</Y>
                        <Z>-1996.83</Z>
                    </POSITION>
                </OBJECT>
            </MAP>
            <GAMESTATE>
                <SELF>
                    <NAME>ProxyBot</NAME>
```

```
                    <POSITION>
                        <X>147.56</X>
                        <Y>-107.68</Y>
                        <Z>-1997.12</Z>
                    </POSITION>
                    <ROTATION>
                        <PITCH>0.0</PITCH>
                        <YAW>3.5739421484051794</YAW>
                        <ROLL>0.0</ROLL>
                    </ROTATION>
                </SELF>
                <PLAYER>
                    <NAME>ProxyBot</NAME>
                    <POSITION>
                        <X>64.0</X>
                        <Y>-118.0</Y>
                        <Z>-1997.1</Z>
                    </POSITION>
                    <ROTATION>
                        <PITCH>0.0</PITCH>
                        <YAW>5.882330835991398</YAW>
                        <ROLL>0.0</ROLL>
                    </ROTATION>
                    <VELOCITY>
                        <X>0.0</X>
                        <Y>0.0</Y>
                        <Z>0.0</Z>
                    </VELOCITY>
                </PLAYER>
                <PLAYER>
                    <NAME>TOM</NAME>
                    <POSITION>
                        <X>-166.47</X>
                        <Y>47.29</Y>
                        <Z>-1996.83</Z>
                    </POSITION>
                    <ROTATION>
                        <PITCH>0.0</PITCH>
                        <YAW>6.007735678927096</YAW>
                        <ROLL>0.0</ROLL>
                    </ROTATION>
                    <VELOCITY>
                        <X>0.0</X>
                        <Y>0.0</Y>
                        <Z>0.0</Z>
                    </VELOCITY>
                </PLAYER>
            </GAMESTATE>
        </CONTEXT>
    </ACTION>
</ACTIONS>
```

Figure 9.
A log file generated by my behavior demonstration tool that includes information about the action performed and its context. The context describes the current map and game state and includes information about objects and characters in the map.

The context associated with each logged action describes the current gamestate and consists of information about the map and characters. Data describing the current gamestate is retrieved from the various WMEs possessed by the ProxyBot. Each object and player in the map is logged with as much information as possible (position, rotation, and velocity are currently supported). This data is extremely important for behavior selection in the case-based reasoning cycle. As described in the introduction, cases consist of a situation, a goal, and a behavior. The context provides information about the situation, the actions provide information about the behavior, and the goal will be described by the user's annotations.

Ontañón et al. created a trace annotation tool called 'WargusIDE' that allows a user to load a log file and annotate their actions. This project uses a plugin module that specifies how the log files (written in XML format) should be parsed. In order to allow users to annotate their traces with WargusIDE, I have created a Mystery Mansion plugin. This plugin consists of a few classes that store the data for each case (MMAction, MMCase, MMContext, MMGameMap, and MMObject), a class that parses the XML file (MMAnnotatedLog), and a class that reads the cases and displays the information in a sensible way to the user (MMActionReader). The classes that store case data are defined in a manner that is directly related to the structure of the log files.

As shown in Figure 9, the most basic element of the log file is <ACTION> which contains <PARAMATERS> and a <CONTEXT> and is parsed into a MMAction object. The MMAction class has data structures that store the following pieces of information: the name of the action, the name of the player who executed the action, the parameter names for the action, and the associated values for each of these parameters.

A <CONTEXT> consists of a <MAP> and a <GAMESTATE> and is parsed into an MMContext object. The MMContext class contains an MMGameMap object which represents the <MAP>, and a list of players that is stored as a list of MMObjects. The MMGameMap class contains a list of MMObjects that holds all of the non-player objects in the map in addition to all of the players. Each of these MMObjects currently has the potential to hold the object's position, rotation, and velocity. However, the user's sensors may not have detected one of these pieces of information, and it will not be saved. In the future, the log file may contain more information about objects and characters such as

their current anger level, sadness level, happiness level, the physical condition of an object (e.g., broken or functional), or a variety of other attribute descriptions.

The MMAnnotatedLog class parses the log file into these objects and creates an MMCase object for each <ACTION> item in the log file. Unfortunately, WargusIDE uses a large amount of code (outside of the plugins) that is specific to the Wargus domain, and it does not properly read my cases. I will continue to work on this project and try to rewrite the domain specific code in WargusIDE so that users may annotate my behavior logs using this tool. Once this is accomplished, I will have enough data to begin converting the logged actions into behaviors that can be executed in UT. After it is possible to execute the behaviors, they will be used for behavior selection during the game to control the non-player characters (NPCs).

Results

I have developed a tool that allows users to demonstrate behaviors in the UT environment and records all of their actions in a log file (see Figure 9 for an example of this result). Multiple users can demonstrate behaviors at the same time, so that the system can learn interactions between characters, and the actions of every character will be written sequentially in the same log file. At the moment, each user must demonstrate behaviors using the same computer because the UnrealScript that controls these characters is not set up to work in network games. Hopefully this will change, and we will allow each user to demonstrate on a separate machine.

Using WargusIDE, demonstrators are able to annotate their log files. However, the current set of goals that can be associated with the actions in the log file is the set from the Wargus domain. If a user wants to annotate the behavior log with Mystery Mansion goals, most of the classes in WargusIDE (and a supporting project, WargusCBR) will have to be rewritten. Once this is finished, annotated log files will be produced that can be used to create a case base for a behavior generation system for the Mystery Mansion project.

Discussion

Once this project has been completed and a system is developed to extract and execute behaviors to control the NPCs in Mystery Mansion, it will be possible to have a highly interactive drama. There will be no need for programmers to hand-write behaviors

for each character. Characters will simply behave according to the manner in which the experts demonstrated actions to the system, and it will be easy to change the entire storyline of the drama and create an entirely different game. If one were using ABL, the behaviors would have to be completely rewritten to accomplish this task.

As long as the case base is large enough so there are multiple cases for each possible scenario, this system will provide interesting, non-repetitive gameplay. The NPCs will be more autonomous because they will effectively 'think' about how they should respond to a situation by remembering past experiences, much like a human would. This approach to behavior generation is far superior to hand-written ABL behaviors that effectively create automatons who behave exactly the same way every time they encounter a specific scenario.

This thesis will hopefully be useful for other areas of research outside the gaming community. Systems that learn behaviors through demonstration can be used in any application that utilizes autonomous agents. The manufacturing industry might want a robot that can learn how to screw any two objects together instead of being programmed to screw one specific bolt into one specific area of an object. This same analogy could be applied to the medical industry for surgeries that require cutting, stitching, and other tasks using autonomous robots. It might be possible to create a system that can write entire movie scripts or make robots that will be able to perform improvisational acting in a theater.

In conclusion, this thesis will provide the base for a larger project that can replace hand-written behaviors with a system that learns behaviors through demonstration. My immediate plan is to complete the work on WargusIDE so that the log files can be annotated. After this is accomplished, I plan to let a few experts in this field evaluate my demonstration tool. If I continue my graduate studies at Georgia Tech, I will try to complete the rest of this system as a graduate thesis.

Acknowledgements

I would like to acknowledge the following people: Ashwin Ram, my thesis advisor, who has provided me with the opportunity to work on the Mystery Mansion project and the necessary hardware and software I needed to do my work; Manish Mehta for constant guidance and help in troubleshooting any problems I had in my research;

Santiago Ontañón for answering any questions I had about relevant literature and brainstorming ideas for my research; the Mystery Mansion project group including: Manish Mehta, Santiago Ontañón, Kinshuk Mishra, and Manu Sharma, who I collaborated with on the storyline; and Colin Smith who wrote ABL behaviors with me during the first semester that I worked on this project.

Works Cited

[1] Agnar, A. and Enric, P. Case-based reasoning: Foundational issues, methodological variations, and system approaches, *Artificial Intelligence Communications, 7(1), March* 1994.

[2] Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A. and Turransky, A. *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, Massachusetts; London, England, 1993.

[3] GamebotsClient.java. Marshall, Andrew N. University of Southern California, Information Science Institute. 2000.

[4] Mateas, M. and Stern, A. Façade: An Experiment in Building a Fully-Realized Interactive Drama.

[5] Mateas, M. and Stern, A. A behavior language for story-based believable agents. *Intelligent Systems*, 17(4), 39-47, 2002.

[6] Ontañón, S., Mishra, K., Sugandh, N. and Ram, A. Case-based planning and execution for real-time strategy games. City, 2007.